Inter-Procedural Resource Sharing in High Level Synthesis through Function Proxies

Marco Minutoli*, Vito Giovanni Castellana†, Antonino Tumeo‡
Pacific Northwest National Laboratory
907 Battelle Blvd, Richland, 99352 WA, USA
*marco.minutoli@pnnl.gov †vitogiovanni.castellana@pnnl.gov
‡antonino.tumeo@pnnl.gov

Fabrizio Ferrandi§
Politecnico di Milano — DEIB
Piazza Leonardo Da Vinci 32
20133, Milan, Italy
§fabrizio.ferrandi@polimi.it

***Abstract*—Modular design is becoming increasingly important in High Level Synthesis (HLS) flows. Current HLS flows generate hierarchical and modular designs that mimic the structure and call graph of input specification translating functions into modules. Function calls are translated instantiating the callee module in the data-path of its caller, allowing for resource sharing when the same function is called multiple times. However, if two different callers invoke the same function, current HLS flows cannot share the instance of the module between the two callers, even if they invoke the function in a mutually exclusive way. In this paper, we propose a methodology that enables sharing of (sub)modules across modules boundaries. Sharing is obtained through *function proxies*, which act as forwarders of function calls in the original specification to shared modules without adding performance penalties. Building on the concept of function proxies, we propose a methodology and the related components to perform HLS of function calls through function pointers, without requiring complete static knowledge of the alias set (point-to set). We show that module sharing through function proxies provides valuable area savings without significant impacts on the execution delays, and that our synthesis approach for function pointers enables dynamic polymorphism.**

I. Introduction

Modular design is becoming increasingly important in HLS flows. By enabling translation of specifications in high-level code (e.g., C) to Register Transfer Level (RTL) specifications in hardware description languages, HLS has the potential to significantly improve designers' productivity when developing custom hardware accelerators for many kernels of time-critical applications. By employing modular and hierarchical approaches, HLS tools can today handle very complex specifications. Modern HLS tools can partition a specification into smaller modules, providing a structured and systematic approach to build the full design. Proper partitioning allows abstracting the implementation details and maintain independence among the modules. Modularity enables the reuse of components (even previously synthesized, and synthesized with other tools) in the design [1], which could be specifically optimized, providing area savings and higher flexibility.

In general, HLS flows generate hierarchical and modular designs that mimic the structure and call graph of the original high-level specification, where a function corresponds to a module. The typical approach is to progressively synthesize functions into modules, navigating the call graph of the original specification from the leaves to the top function. A function call corresponds to the instantiation of the related module into the data-path generated for the caller. If a caller invokes the same function multiple times, the HLS flows can usually generate a data path that reuses the same instance of a module for the different calls, depending on resource/performance trade-offs. However, if two different callers invoke the same function, current HLS flows do not share the instance of the module between the two callers, even if they invoke the function in a mutually exclusive way. With minor variations, this is the strategy implemented both in research tools (e.g., LegUp [2], PandA [3], Shang [4], GCC2Verilog [5]) and in commercial products (e.g., Vivado HLS from Xilinx [6], AutoPilot from AutoESL, Synphony from Synopsys [7]). Moderns application may have complex call graphs, where the same function is invoked from very different paths, and such an approach is clearly sub-optimal in terms of area.

In this paper, we propose a methodology that supports the sharing of (sub)modules across modules boundaries. Sharing is obtained through *function proxies*, which act as forwarders of function calls in the original specification to shared modules without adding performance penalties. Building on the abstraction of function proxies, we propose a methodology and the related components to perform HLS of function calls through function pointers, without requiring complete static knowledge of the alias set (point-to set). This enables performing calls from the same call points in a data path to different (shared) modules depending on dynamic conditions. In other words, the approach enables the synthesis of specifications that employ dynamic polymorphism.

The main contributions of this paper are:

- The introduction of *function proxies* as an architectural components for modular HLS;
- a methodology for the HLS of shared (sub)modules across module boundaries employing function proxies;
- a methodology that enable synthesis of function pointers.

The remainder of this paper is organized as follows. Section II introduces the concept of function proxies, describing the architecture and the notification mechanisms. Section III explains how function proxies enable the synthesis of function pointers. Section IV presents the synthesis results, highlighting

the benefits of module sharing across module boundaries. Section V compares our approach to the related work. Finally, Section VI concludes the paper.

II. Methodology

Modern HLS tools automatically generate the hardware implementation of behavioral specifications commonly described through programming languages such as C/C++. The result of the synthesis is a HDL (Verilog, VHDL) description of the design, typically implemented as Finite State Machine with Data-path (FSMD) modules. When synthesizing applications characterized by complex call structures the generated designs typically are modular, and the design hierarchy mimics the structure of the specification's Call Graph (CG). The synthesis proceeds one function at a time, starting from the leaves of the CG: the tool embeds FSMDs of callee functions into the data-path of the caller functions. Figure Figure 1a proposes an example CG, while Figure Figure 1b shows the corresponding modular architecture, obtained through typical HLS techniques. While preserving modularity, this approach requires the allocation of at least one callee function module for each caller module. In the proposed example, *funC* is allocated within both *funA* and *funB* modules. This issue occurs regardless of the schedules of the callers: conventional HLS flows allocate multiple callee modules also when they execute in mutual exclusion (e.g., when *funB* depends on the call of *funC* within *funA*). This leads to sub-optimal resource utilization, which may be particularly significant for complex applications.

A common technique that may improve resource utilization is function inlining. Function inlining provides several valuable opportunities for optimization:

- it may improve Instruction Level Parallelism exploitation, leading to latency improvements;
- by removing function boundaries, it may improve code transformation optimizations, such as constant propagation and dead code elimination;
- it allows sharing at the level of functional units.

However, depending on the application characteristics, when employing inlining the following issues may arise:

- an excessive flattening of the call structure may dramatically increase the number of operations within a function, potentially affecting the controllers complexity. As controller complexity grows, the synthesis tools start introducing long wires and high load capacitances. This, in turn, increases the path lengths between the controller and the data-path selectors, degrading the overall performance;
- control and data dependencies may still reduce the amount of extractable ILP; in such a case, fine-grained sharing of functional units leads to worse latency/area trade-offs when compared to sharing at coarser granularities. Sharing at the level of FUs in fact, requires allocation of steering logic for each shared FU; sharing a function module instead, only requires allocation of steering logic for its inputs.

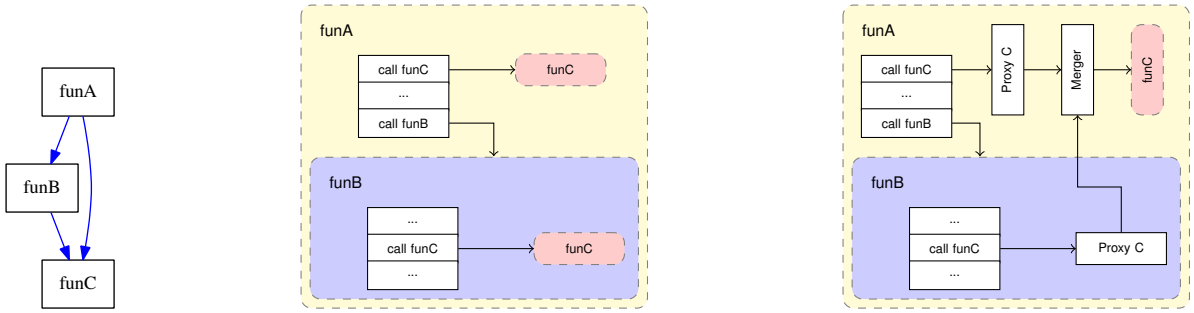
To overcome limitations in resource sharing of conventional HLS flows, we propose a general methodology that allows sharing of modules across data-path boundaries, at every level of the design hierarchy. Our approach is based on the definition of lightweight control elements, called *function proxies*, which enable the management of shared resources across the design hierarchy, without affecting the design and the complexity of the controllers. The technique is orthogonal to inlining: with complex CGs inlining could actually improve the quality of the results by providing larger candidates for sharing.

A. Function Proxy Architecture

As in typical modular design techniques, we model function modules as custom synthesized units that expose a simple interface. The interface includes: start and done signals, input parameters, and return values. The controller of the caller manages the function module, while the caller data-path embeds the module itself. In our design, we preserve this simple, yet effective, structure, taking advantage of function proxies. Function proxies *substitute* the instance of a shared module in the caller data-path, redirecting control and data signals to the proper module instance. This does not affect the behavior of the controller of the caller. The generated architecture includes, for every shared function module, a single instance of the module, allocated within the data-path of the caller at the higher level in the design hierarchy, and a proxy for each call of the function, embedded in the data-path of each caller module. The signal propagation mechanism exploits a custom hardware component, called *merger*, which is associated with the instantiated shared resource. The merger collects signals coming from the proxies and forwards them to the instance of the module. It also collects the outputs of the shared module and forwards them to the function proxies. In our architecture, only one proxy is active at a time, meaning that calls to shared modules always occur in mutual exclusion. While inactive, a proxy always outputs *null* valued signals. This simplifies the design of the merger, which can be efficiently implemented by OR-ing the incoming signals and broadcasting back the outputs to the connected proxies. Inactive proxies just ignore the incoming signals (done and return values). Figure Figure 1c schematizes the architecture associated with the CG of Figure Figure 1a, obtained by enabling the sharing of *funC*. The shared instance of *funC* is allocated within *funA* module. Dedicated proxies manage calls of *funC* in both *funA* and *funB*.

B. High-Level Synthesis of function proxies

To enable sharing of function modules during HLS, we propose an algorithm for the identification and selection of suitable candidates. Our algorithm identifies candidates by analyzing the CG of the specification: the algorithm marks functions with more than one caller as *sharable* among module boundaries. Nevertheless, not all the candidates are selected for sharing. In fact, sharing small function modules may provide limited area reductions because proxies, mergers, and steering logic necessarily induce an area overhead that may



(a) Call graph (b) Without Functin proxy (c) With Functin proxy

Figure 1: Architecture obtained before and after function proxy introduction.

compensate any benefits. This may happen even if we designed proxy and merger components trying to minimize resource requirements. For this reason, we devised a simple heuristic for the selection algorithm. The procedure computes the schedule of the candidate functions: if the execution latency of a function is constant (e.g., its latency is not input-dependent, or it does not include variable latency operations) and limited in terms of clock cycles, it is not selected for sharing. We choose latency as selection criteria for two reasons. First, low-latency modules typically small enough in terms of area to limit the benefits of sharing. Second, in presence of function level parallelism, calls to modules with constant latency are easy to schedule in parallel. This reduces the overall latency of the design, but requires to instantiate multiple modules. We have implemented the methodology in a publicly available HLS framework and, after an empirical design space exploration we have found a threshold latency of 4 clock-cycles to provide the best overall area/performance trade-offs. After the selection phase, the module allocation identifies the data path entity in the design hierarchy where to embed the function module instances. The algorithm once again analyzes the CG of the application, and chooses the proper level in the hierarchy by identifying the *dominator* of the calls to the shared function. For complex CGs, the dominator function may not include any call to shared functions.

III. Synthesis of function pointers

The introduction of function proxies enables sharing a single instance of a module that implements a function. By exploiting this feature, it is possible to extend the methodology to support the synthesis of applications with function pointers. State of the art approaches commonly deal with function pointers by:

- static resolution of pointers: the frontend compiler can generate specialized versions of synthesized functions, if it can statically resolve function pointers to a single candidate. This is the case of calling a function that takes as input a function pointer, and passing a function name as an argument;
- statically computing the alias set of the function pointers: by performing alias analysis, it is possible to translate calls that use function pointers into a switch block that covers all the possible alternatives.

```
int laplacian(char *, char *, int, int);
int make_inverse_image(char *, char *, int, int);
int sharpen(char *, char *, int, int);
int sobel(char *, char *, int, int);

int (*pipeline[MAX_DEPTH])(char *, char *, int, int);

void UserApp(char *in, char *out, int x_size, int y_size) {
  // ...
  // Pipeline configuration using function pointers
  add_filter(0, make_inverse_image);
  add_filter(1, sharpen);
  // ...
  // execute is synthesized in hardware
  execute(in, out, x_size, y_size);
}
void execute(char *in, char *out, int x_size, int y_size) {
  int i = 0;
  for (i = 0; i < MAX_PIPELINE_DEPTH; i++) {
    if (pipeline[i] == 0) break;
    // here other hw accelerator are called
    // using function pointers
    int res = pipeline[i](in, out, x_size, y_size);
    if (res != 0) return;
    swap(in, out);
  }
  move_if_odd(i, in, out);
}
```

Listing 1: Image processing application using function pointers.

The underlying assumption of these two approaches is that the HLS flow has the complete knowledge of the point-to set of a function pointer during the synthesis. Having a single instance of a function, through *function proxies*, enables a more dynamic support for calls through function pointers. Our methodology supports synthesis of function pointers by extending modules with a memory mapped interface and by introducing a communication protocol that implements the call mechanism through such an interface.

Memory mapped interfaces are a well established design pattern that is widely used for communication with hardware accelerators and peripherals. The following sections discuss how a memory mapped interface allows implementing a call mechanism for function pointers in a simple and effective way.

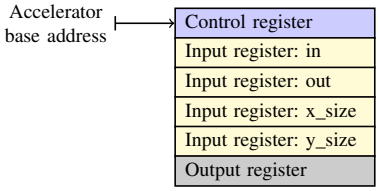


Figure 2: Memory mapped interface of filters: `laplacian`, `make_inverse_image`, `sharpen` and `sobel`.

1) Memory mapped interface: The proposed flow automatically generates the memory mapped interface according to the prototype of the synthesized function. The interface includes a control register, a set of input registers and an output register.

During the memory allocation step, the HLS flow assigns a unique ID to each function. The ID works as the function base base address and is associated with the control register. At the same time, the memory allocation step reserves addresses for the function parameters and for the function return value. Following this schema, the control register is the first element allocated in the address space of an accelerator, followed by the memory mapped register of the function interface. This solution allows associating the base address of a function (its function pointer) with the address of its control register in the synthesized architecture.

The control register gives access to the internal state of the accelerator. The state can be: idle, computing, done. The main purpose of the control register is to enable other processing elements to start the computation and to identify when the computation completes.

Input registers are allocated next to the control register. They store the value of input parameters defined in the function prototype of the synthesized accelerator. The last element of the module interface is the output register, which stores the return value of the synthesized function. Registers for input parameters and return value are generated only if needed. For example, modules synthesized from function returning `void` do not have the output register. Similarly, modules synthesized from functions without input parameters do not have input registers. Figure Listing 1 shows an application that implements a dynamically (re)configurable image processing pipeline. The application contains a set of four filters: *laplacian, make_inverse_image, sharpen and sobel*. The pipeline is modeled through an array of function pointers (Listing 1 line 6). Figure Figure 2 shows how the proposed flow translates the prototypes of the filters into the memory mapped interface.

2) Indirect call mechanism: The function call mechanism for function pointers directly derives from the module interface definition. Each caller performs three operations to invoke a module through a function pointer. First, it writes parameters into the input registers of the memory mapped interface. Then, it starts the computation of the callee by writing in the callee control register, and waits for the result. Finally, when the callee completes, it reads the return value from the callee

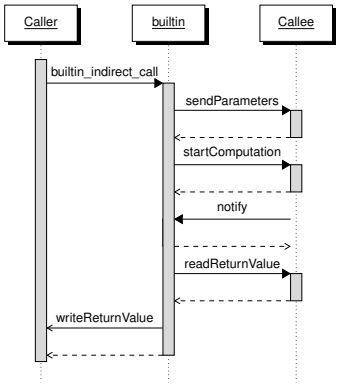


Figure 3: Sequence diagram of an indirect function call.

interface and continues its computation. The mechanism is analogous to software function calls. The two steps performed are equivalent to filling the activation record of the function, and then jumping to the address of the first instruction of the called function.

The layout of the memory interface is the same for all the functions with the same signature because it is generated according to the function prototype. This property guaranties the generality of the call mechanism.

The value of the function pointer controls which module is called, while the standard layout of the memory mapped interfaces allows accessing the registers only using their relative addresses.

3) Notification mechanism: The status register stores the current state of the module accelerator. Using this information, a caller can periodically poll the control register of the invoked module to find out when it completes. Unfortunately, this strategy does not scale with the number of hardware modules concurrently active. In fact, periodical polls can congest the bus. Furthermore, the unnecessary bus traffic can prevent called modules to perform memory operations needed to complete their computation, resulting in deadlocks. To avoid this issue, our methodology includes an asynchronous notification mechanism that is activated during the call process.

A memory operation is, again, at the core of the mechanism. The HLS flow associates a unique ID, named *notification address*, to every call site. So, on one side, the *notification address* is stored in the control register during the call. On the other side, the called module employs the address stored in its control register to notify the caller of its termination. The notification only requires a write operation to the address stored in the control register. Once the caller intercepts this store, the computation restarts by reading the return value from the memory mapped interface of the called module (Figure Figure 3).

Allocating different addresses for each indirect call site allows distinguishing among calls to the same module from different callers.

4) Function pointer run-time cost: The indirect call mechanism introduces a performance penalty over the direct call mechanism. Assuming a single bus architecture, we can com-

```
void execute(char *in, char *out, int x_size, int y_size) {
  int i = 0;
  for (i = 0; i < MAX_PIPELINE_DEPTH; i++) {
    if (pipeline[i] == 0) break;
    // here other hw accelerator are called
    // using function pointers
    __builtin_indirect_call(
        pipeline[i], 1, in, out, x_size, y_size, &res);
    if (res != 0) return;
    swap(in, out);
  }
  move_if_odd(i, in, out);
}
```

Listing 2: Image processing example after builtin insertion.

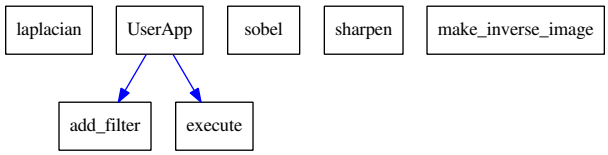


Figure 4: Initial call graph of Listing 1.

pute the overhead (in clock cycles) of the call mechanism (CO) as:

$$CO = W_l(N_p + 1) + lhs(W_l + R_l) \quad (1)$$

where N_p is the number of function parameters, W_l and R_l are respectively write and read latency of the memory mapped registers and lhs is 1 when the call instruction is on the left hand side of an assignment or 0 otherwise. The first term takes into account the run-time cost for parameter passing plus the start command, while the second term accounts for the time spent to retrieve the returned value. Parallelizing parameter passing to the called module with a multi-channel architecture can reduce the cost of the mechanism in terms of latency.

5) Source code transformations: We added two transformation passes to the HLS flow to support the methodology described in the previous section. The first transformation substitutes function calls performed through function pointers with calls to a builtin function (the transformed code of the initial image processing example is shown in Figure Listing 2). The builtin function is defined as a variadic function. Its first argument is the function pointer of the function to be called. The second argument is a Boolean flag, which is true when the last passed parameter is the address of the return value, or false otherwise. Further arguments are the list of input parameters passed to the function and the optional returning value address. The HLS flow recognizes the builtin function and translates it in a module implementing the indirect call mechanism.

Once the the first transformation inserts the builtins, the second transformation modifies the application call graph by adding edges that connects callers using pointers to functions matching the pointer type. This transformation retrieves the missing information about function calls using function pointers. Figures Figure 4 and Figure 5 show how the transformation modifies the call graph of the application in Figure Listing 1.

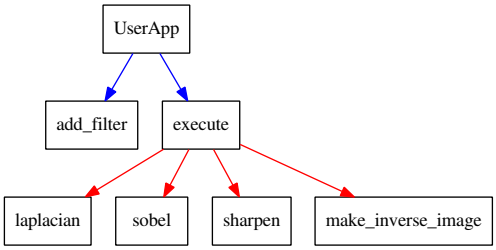


Figure 5: Enriched call graph.

IV. Experimental evaluation

We validated our approach by extending Bambu, the HLS tool of the open-source PandA [3] framework, freely downloadable from the internet. The tool takes in input applications described in C and generates their Verilog implementations. It interfaces with the GCC compiler as a front-end, which allow exploiting several optimization techniques typical of software compilation, that are also profitable in HLS. These for example include hoisting, constant propagation, dead code elimination, function inlining, loop unrolling. We simulated the resulting designs with ModelSim 10.3, and synthesized them with Vivado 2014.4, targeting a Xilinx Virtex-7 device (package xc7vx485t). We targeted a frequency of 100 MHz for the synthesis, and show the estimated maximum frequency for the synthesized circuits. We characterize area requirements by reporting the number of Look Up Tables (LUTs), Flip Flops (FFs), and Digital Signal Processing (DSP) units post place and route. We evaluate the effects of the introduction of function proxies into the synthesis flow, and then provide a validation for the synthesis of function pointers.

A. Function proxies

We validated our sharing approach by synthesizing the CHStone benchmark suite [8], a reference benchmark for HLS tools. CHStone benchmarks have very simple call graphs, so the front-end compiler, in most cases, completely flattens the designs by inlining all the functions, thus making our methodology ineffective. In these settings our sharing strategy can be applied only on the *mpeg2* benchmark. On this benchmark, the function proxies reduce LUTs and FFs utilization by more than 50% (from 4425 to 2035 LUTs, from 3367 to 1429 FFs, 0 DSPs). In order to highlight the effects of the proposed techniques, we have selectively disabled inlining for functions with more than one caller. We underline that our methodology does not have the intent of replacing other optimization techniques (such as inlining), but should rather be used in combination with them. Table Table I shows the obtained synthesis results., Overall, enabling sharing across data-path boundaries reduces LUTs utilization by 17%, FF utilization by 18.85%, DSP utilization by 1.77%, on average. The benefits of the proposed methodology widely vary according to the benchmark characteristics For benchmarks featuring small functions or limited opportunities for sharing we report a gain in area utilization lower than 4%; for benchmarks with

Table I: CHStone benchmarks with (FP) and without ($noFP$) function proxies disabling inlining.

	#Cycles	Freq. (MHz)		#LUTs		LUT Gain	#FFs		FF Gain	#DSPs		DSPs Gain
		FP	noFP	FP	noFP		FP	noFP		FP	noFP	
adpcm	17996	94.10	94.94	5595	6143	8.92%	4340	4564	4.91%	63	69	8.70%
aes	1906	128.22	124.30	5984	7751	22.80%	5342	7141	25.19%	0	0	0.00%
blowfish	88763	114.39	106.95	3436	6241	44.94%	2185	3699	40.93%	0	0	0.00%
dfadd	520	133.89	139.16	3004	3937	23.70%	2303	2772	16.92%	0	0	0.00%
dfdiv	1152	112.51	109.89	5781	5783	0.03%	4219	4217	−0.05%	32	32	0.00%
dfmul	143	111.91	114.05	1643	1672	1.73%	983	983	0.00%	16	16	0.00%
dfsin	34182	104.22	102.89	13917	16423	15.26%	10151	12261	17.21%	51	51	0.00%
gsm	2920	91.93	87.97	4279	4309	0.70%	3619	3657	1.04%	25	25	0.00%
jpeg	571065	97.66	95.07	12642	12717	0.59%	6854	7238	5.31%	7	8	12.50%
mpeg2	4235	115.34	113.83	2362	5105	53.73%	1886	4031	53.21%	0	0	0.00%
sha	87362	155.38	127.06	3338	5442	38.66%	3811	6643	42.63%	0	0	0.00%
Averange		114.50	110.56			19.19%			18.85%			1.77%

Table II: libm function subset and basic_math with (FP) and without ($noFP$) function proxies.

	#Cycles	Freq. (MHz)		#LUTs		LUTs Gain	#FFs		FFs Gain	#DSPs		DSPs Gain
		FP	noFP	FP	noFP		FP	noFP		FP	noFP	
acoshf	430	93.06	91.01	7353	12667	41.95%	7397	11943	38.06%	13	39	66.67%
asinhf	313	93.56	90.35	7535	12009	37.26%	7491	11381	34.18%	13	39	66.67%
atan2f	366	94.95	89.76	4031	6011	32.94%	3879	5856	33.76%	13	24	45.83%
atanhf	512	91.31	92.13	5571	7496	25.68%	5108	7306	30.08%	13	26	50.00%
cosf	213	108.24	115.19	10866	17405	37.57%	10372	16464	37.00%	2	12	83.33%
coshf	243	91.89	91.16	6988	11296	38.14%	6578	10643	38.19%	13	41	68.29%
erfcf	273	91.84	90.24	7543	10314	26.87%	7817	10338	24.39%	13	28	53.57%
erff	224	91.39	89.69	7447	10240	27.28%	7786	10316	24.53%	13	28	53.57%
expf	204	92.81	93.27	3807	3946	3.52%	3620	3746	3.36%	13	15	13.33%
gammaf	530	92.76	90.03	11671	18516	36.97%	12424	18878	34.19%	13	32	59.38%
hypotf	176	134.92	129.07	3773	3893	3.08%	3744	3871	3.28%	2	4	50.00%
lgammaf	773	90.01	91.31	11888	18678	36.35%	12385	18858	34.32%	13	32	59.38%
powf	1050	93.16	91.24	8530	8519	−0.13%	7691	7833	1.81%	13	15	13.33%
sinf	139	106.94	108.18	11078	17389	36.29%	10367	16464	37.03%	2	12	83.33%
sinhf	576	92.51	90.46	7108	11944	40.49%	6623	11395	41.88%	13	41	68.29%
tanf	481	92.27	92.39	12591	17156	26.61%	11043	15747	29.87%	13	21	38.10%
tanhf	636	93.69	91.92	5514	7567	27.13%	5269	7341	28.23%	13	26	50.00%
basic_math	25316299	84.27	89.69	52061	148075	64.84%	28982	87166	66.75%	58	250	76.80%
Averange		95.97	95.39			30.16%			30.05%			55.55%

more complex call structures, we report much more valuable area reductions, up to over 50% (for both LUTs and FFs) for *mpeg2*. Simulation results confirm that the introduction of the additional components (proxies and mergers) for supporting the sharing do not lead to any penalty in terms of number of clock cycles needed for execution. Although such components, being completely combinational, may slightly increase the length of critical paths, we do not report any significant reductions of the achievable frequency. In several examples (e.g. *sha*), we even obtain frequency improvements: this may be associated to the effects of the area reduction during the synthesis through Vivado.

We evaluate our methodology also on applications featuring more complex call structures. We selected a subset of 17 *libm* primitives from Newlib [9] and the basic_math benchmark from the mibench suite[10]. For these kernels, the front-end compiler does not flatten the specifications through inlining, due to the complexity of the call graphs. In particular, *basic_math* invokes the same functions from many different paths of its call graph, highlighting the effectiveness of our approach (over 60% improvement in both LUTs and FFs).

Table III: Image filtering application.

	#Cycles	#LUT Pairs	#DSPs	#BRAMs	Freq. (MHz)
$test_0$	993104	30667	46	6	107.25
$test_1$	1026151	30667	46	6	107.25

Table Table II shows that the introduction of function proxies reduces the number of LUTs by 30.16%, FFs by 30.05%, DSPs by 55.55%, on average. The benchmarks that present the smallest area reductions (*expf*, *hypot*, and *pow*) are characterized by small functions, whose sharing is less profitable. Also in this case, we do not report any latency penalty in terms of both clock cycles to complete execution and achievable frequency.

B. Function pointers

We validated the support for the synthesis of function pointers with yet another set of benchmarks, since CHStone and

Table IV: Sorting/searching algorithms using function pointers.

	#Cycles	#LUT Pairs	Freq. (MHz)	#DSPs
bsearch-glibc	1292	6782	101.77	0
bsearch-musl	1301	6770	121.23	0
bsearch-newlib	1292	6836	104.19	0
bsearch-uclibc	1292	6936	105.55	0
qsort-glibc	2285721	10792	113.15	1
qsort-uclibc	2683672	10925	101.07	0

libm do not include any benchmark using function pointers. For the validation, we selected several non-recursive qsort and bsearch functions from various open-source C standard libraries (glibc, uclibc, newlib, musl), [9], [11]–[13], and an implementation of the image processing applications used to illustrate our approach in Section Listing 1. All the selected searching and sorting algorithms operate over an array of 1000 structures each containing 10 floating point numbers. The input data has been randomly generated. Table IV presents simulation (execution latencies in terms of clock cycles) and synthesis (number of FF/LUT pairs and DSPs) results for the considered applications. We verified that the results obtained by simulating the circuit generated by our HLS flow are correct and provide the same results of the software implementation when processing the same input data. We underline that without support for function pointers, a developer would have to rewrite the software implementation of these kernels to remove them. Instead, our flow allows to directly synthesize them without any modification.

Finally, Table Table III reports the simulation and synthesis results for the image filtering application operating on an image of 64×64 pixels (purposely small to reduce simulation times.). We validated two configurations: in the first, the pipeline includes only the sharpening filter ($test_0$), while in the second we also added the inverse image computation ($test_1$).

V. Related Work

Many works have looked at improving resource utilization in HLS at different levels of the design flow. Techniques that restructure the input specification include clustering, data-path fusion and pattern matching[8], [14]–[17]. Compared to such solutions, our work looks at improving resource utilization at a coarser granularity, but can exploit these finer grain techniques to increase its effectiveness.

[15], [18] introduce procedure exlining. The technique aims at eliminating redundancies from the input specification through restructuring. The papers present a semi-automated method that combine approximate matching and regular expression with other heuristics in order to produce a better function partitioning of the original specification.

[8], [16] propose an Integer Linear Programming (ILP) formulation that searches for the optimum balance between inlining and exiling. The methodology has been also extended with clustering techniques to merge similar functions.

These works have the final effect of improving the synthesis results by promoting function reuse. Those techniques improve the modularity of the specifications, possibly making the adoption of our sharing methodology more effective.

[17] presents a pattern-based methodology that improve resource utilization for Field Programmable Gate Array (FPGA) devices. Its main goal is to find similar patters of operations and, when possible, reduce them to a common form to enable sharing of block of functional units, lowering the number of multiplexer introduced in the design. Nevertheless, the approach only works at the module level and does not allow inter-module sharing of functional units. Our approach, instead, overcomes this limitation by employing function proxies, which enable sharing blocks of common operations (whole functions) across the boundaries of modules.

The idea of globally sharing module instances in the synthesized architecture has been already explored in the past. The Handel-C [19] language provides constructs called *shared expressions* for this purpose. Shared expressions allow sharing of resources across different sections of the synthesized specification. However, it is designer responsibility to identify shareable operations and define accordingly the shared expressions. Our approach, instead, is a fully automated HLS methodology that makes inter-modular resource sharing completely transparent to the designer. Furthermore, it operates at higher granularity than shared expressions (i.e., function level rather than operations).

The work in [20], [21] suggests an alternative approach for the synthesis of function pointers. The paper propose to synthesize calls through function pointers by generating control blocks assign values to function pointers to resolve the call. The generated control block contains a branch for each element in the alias set of a function pointer. This implies that the alias set of the function pointers must be known at compile time.

Our methodology overcomes this limitation. The indirect call mechanism defines a communication protocol and works without the knowledge of the alias set of the function pointer. This enables modeling with function pointers externally developed Intellectual Properties (IPs), unavailable during the synthesis.

The work in [22] proposes a methodology that supports function pointers and recursion through stream rewriting, a different model of computation. The different underlying model make a direct comparison with our approach unfeasible.

The methodology in [5], [23] proposes a solution for HW/SW cross calls. The work defines an architecture and the related mechanism that allows calls from SW to HW, from HW to SW and from HW to HW (including recursive calls). In particular, the architecture proposed in [23] controls the execution of the HW IPs by introducing a hardware controller that stores parameters and starts the computation of hardware accelerators. This centralized controller simplifies the communication between HW and SW, but it does not allow concurrent execution of two or more IPs. Our architecture does not present this limitation because it uses a distributed

control mechanism. From this point of view, our approach paves the way to the synthesis of parallel applications based, for example, on the *pthreads* library.

[24] presents a solution that allows expressing *static* (compile time) polymorphic behavior through SystemC and C++ template meta-programming. Our approach based on function proxies, which enables synthesis of function pointers, enables *dynamic* polymorphism.

VI. Conclusion

In this paper, we proposed a methodology for modular HLS which enables sharing (sub)modules across module boundaries. In conventional HLS flows, modules synthesized from functions cannot be shared across different callers. Our approach, which employs the abstraction of function proxies, instead, allows sharing a single instance of a function module in the whole synthesized architecture. We detailed the architecture of function proxies, how they are translated to a memory mapped interface, and discussed how they enable the synthesis of function pointers. We have validated the methodology and demonstrated that, for applications with a sufficiently complex call graphs, it can provide valuable savings in area without incurring in significant performance overheads. Possible future extensions include the synthesis of recursive functions and the synthesis of accelerators from parallel specifications without the need of external (hardware or software) coordinators to launch hardware threads. Parallel specifications could employ the *pthreads* library, which directly invokes threads as function pointers, or OpenMP annotations, which normally gets translated into thread invocation through function pointers.

References

[1] A. A. Jerraya, H. Ding, P. Kission, and M. Rahmouni, *Behavioral Synthesis and Component Reuse with VHDL*. Springer US, 1997, ISBN: 978-1-4613-7899-0.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems", *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, 24:1–24:27, Sep. 2013, ISSN: 1539-9087.

[3] (2004). Panda: a framework for hardware-software co-design of embedded systems, [Online]. Available: http://panda.dei.polimi.it.

[4] (2014). The shang high-level synthesis framework, [Online]. Available: https://github.com/OpenEDA/Shang.

[5] G. N. T. Huong and S. W. Kim, "GCC2Verilog compiler toolset for complete translation of C programming language into Verilog HDL", *ETRI Journal*, vol. 33, no. 5, pp. 731–740, 2011.

[6] T. Feist, "Vivado design suite", *Xilinx, White Paper Version*, vol. 1, 2012.

[7] (2014). Synphony model compiler, [Online]. Available: http://www.synopsys.com/systems/blockDesign/HLS/Pages/default.aspx.

[8] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis", *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[9] (2014). Newlib, [Online]. Available: https://sourceware.org/newlib/.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: a free, commercially representative embedded benchmark suite", in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, IEEE, 2001, pp. 3–14.

[11] (2014). Glibc, [Online]. Available: http://www.gnu.org/software/libc/.

[12] (2014). Uclibc, [Online]. Available: http://www.uclibc.org.

[13] (2014). Musl libc, [Online]. Available: http://www.musl-libc.org/.

[14] F. Vahid and D. D. Gajski, "Clustering for improved system-level functional partitioning", in *System Synthesis, 1995., Proceedings of the Eighth International Symposium on*, IEEE, 1995, pp. 28–33.

[15] F. Vahid, "Partitioning sequential programs for CAD using a three-step approach", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 3, pp. 413–429, 2002.

[16] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Function call optimization for efficient behavioral synthesis", *IEICE Transactions*, vol. 90-A, no. 9, pp. 2032–2036, 2007. DOI: 10.1093/ietfec/e90-a.9.2032. [Online]. Available: http://dx.doi.org/10.1093/ietfec/e90-a.9.2032.

[17] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction", in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '08, Monterey, California, USA: ACM, 2008, pp. 107–116, ISBN: 978-1-59593-934-0.

[18] F. Vahid, "Procedure exlining: a transformation for improved system and behavioral synthesis", in *Proceedings of the 8th international symposium on System synthesis*, ACM, 1995, pp. 84–89.

[19] (2014). Handel-c synthesis methodology, [Online]. Available: http://www.mentor.com/products/fpga/handel-c.

[20] L. Séméria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 6, pp. 743–756, 2001.

[21] L. Séméria and G. De Micheli, "Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 2, pp. 213–233, 2001.

[22] L. Middendorf, C. Bobda, and C. Haubelt, "Hardware synthesis of recursive functions through partial stream rewriting", in *Proceedings of the 49th Annual Design Automation Conference*, ACM, 2012, pp. 1207–1215.

[23] G. Nguyen Thi Huong, Y. Na, and S. W. Kim, "Applying frame layout to hardware design in FPGA for seamless support of cross calls in CPU-FPGA coupling architecture", *Microprocessors and Microsystems*, vol. 35, no. 5, pp. 462–472, 2011.

[24] T. R. Muck and A. A. Frohlich, "Towards unified design of hardware and software components using c++", *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2013, ISSN: 0018-9340. DOI: http://doi.ieeecomputersociety.org/10.1109/TC.2013.159.